THE DECOMPOSITION OF AN ARBITRARY THREE-DIMENSIONAL
PLANAR POLYGON INTO A SET OF CONVEX POLYGONS(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA J H POTTS DEC 86
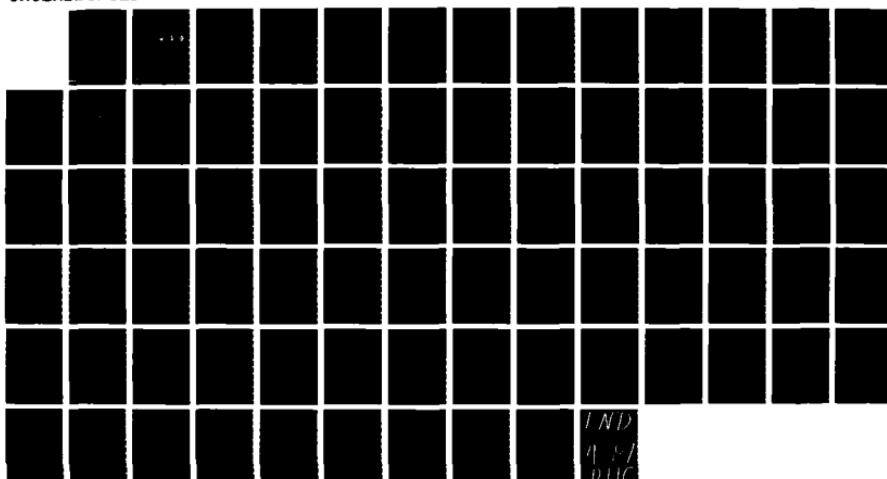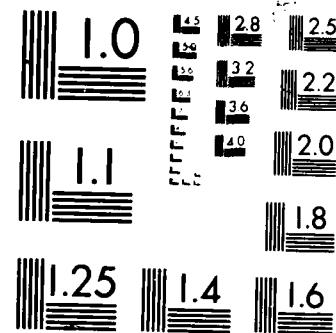
MICROCOPY RESOLUTION TEST CHART

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DECOMPOSITION
OF AN ARBITRARY THREE-DIMENSIONAL
PLANAR POLYGON INTO A
SET OF CONVEX POLYGONS

by

Jeffrey H. Potts

December 1986

Thesis Advisor: Michael J. Zyda

87 3   10 001

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION unclassified		1b RESTRICTIVE MARKINGS			
2a SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.			
2b DECLASSIFICATION/DOWNGRADING SCHEDULE					
4 PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)			
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable) 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School			
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000			
8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8c ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification) THE DECOMPOSITION OF AN ARBITRARY THREE-DIMENSIONAL PLANAR POLYGON INTO A SET OF CONVEX POLYGONS

12 PERSONAL AUTHOR(S) Potts, Jeffrey H.

13a TYPE OF REPORT Master's Thesis	13b TIME COVERED FROM ___ TO ___	14 DATE OF REPORT (Year, Month, Day) 1986 December	15 PAGE COUNT 77

16 SUPPLEMENTARY NOTATION

17	COSATI CODES		18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer graphics; polygons; polygon decomposition; polygon concavity.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

We present in this study a three step algorithm for the decomposition of arbitrary, three-dimensional, planar polygons into convex polygons. Through a series of translations and rotations, an arbitrary polygon is mapped onto the x-y plane, then broken into a set of convex polygons, and finally mapped back to the polygon's original coordinate system for filling and display by special graphics hardware.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Prof Michael Zyda	22b TELEPHONE (Include Area Code) (408) 646-2305	22c OFFICE SYMBOL Code 52Zk

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted All other editions are obsolete

The Decomposition
Of An Arbitrary Three-Dimensional Planar Polygon
Into A Set Of Convex Polygons

by

Jeffrey Hal Potts
Captain, United States Marine Corps
B. S.,United States Naval Academy, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1986

Author: _____
Jeffrey H. Potts

Approved by: _____
Michael J. Zyda, Thesis Advisor

_____
Robert B. McGhee, Second Reader

_____
Vincent Y. Lum, Chairman,
Department of Computer Science

_____
Kneale T. Marshall,
Dean of Information and Policy Sciences

2

ABSTRACT

We present in this study a three step algorithm for the decomposition of arbitrary, three-dimensional, planar polygons into convex polygons. Through a series of translations and rotations, an arbitrary polygon is mapped onto the x-y plane, then broken into a set of convex polygons, and finally mapped back to the polygon's original coordinate system for filling and display by special graphics hardware.

Accesion For		
NTIS CRA&I		☑
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

QUALITY INSPECTED 1

3

TABLE OF CONTENTS

4

I. INTRODUCTION

Computer hardware advancements applied to the field of computer graphics have recently made possible the real-time display and animation of three-dimensional object models. In most currently produced graphics systems, the three-dimensional (3D) object models are comprised of sets of 3D polygons. For years the key bottleneck to generating real-time images of 3D models has been the limited speed with which the 3D polygons comprising the model have been filled. This bottleneck has been alleviated recently with the advent of special hardware designed to fill polygons at rates exceeding 44 million pixels per second [Ref. 1]. For real-time animation of 3D object models, such polygon fill rates are essential.

The majority of the graphics systems that provide polygon fill hardware only provide hardware for filling convex polygons. Concave polygons passed to such hardware usually are filled incorrectly, if at all. To utilize the capabilities of such hardware for the general polygon case, that is, a mix of concave and convex polygons, it is necessary to have some means of decomposing an arbitrary 3D planar polygon into a set of 3D planar convex polygons completely covering the area defined by the original polygon. The following study attempts to provide such an algorithm. We then discuss the implementation and performance characteristics of that algorithm on the Silicon Graphics, Inc. IRIS workstation.

6

II. BACKGROUND

A. DEFINITIONS

A polygon is defined as a geometric figure that is described by an array of x. y. and z coordinates for its vertices. A two dimensional polygon lying in the x-y plane is a special case of a three dimensional polygon in which the z coordinates are all zero. A convex polygon is one that has no points of negative curvature along its boundary [Ref. 2: p. 217]. This means that a line can be drawn from any point inside the polygon to any of its vertices and have the line lie entirely within the polygon boundary [Ref. 3: p. 350]. A concave polygon fails the above test. i.e. we cannot draw a line from every point inside the polygon to every vertex and have the line remain within the polygon boundary. However. any point inside a concave polygon can be shown to lie inside a convex polygon created by some of the vertices and sides of the original polygon [Ref. 3: p. 330]. Figure 2.1 provides examples of convex and concave polygons.

B. DECOMPOSITION

To fill a concave polygon utilizing hardware that only correctly fills convex polygons. we must first break down the polygon into simpler convex polygons that together cover the area of the original. The process of decomposing a concave polygon does not necessarily yield a single unique set of convex polygons. but the
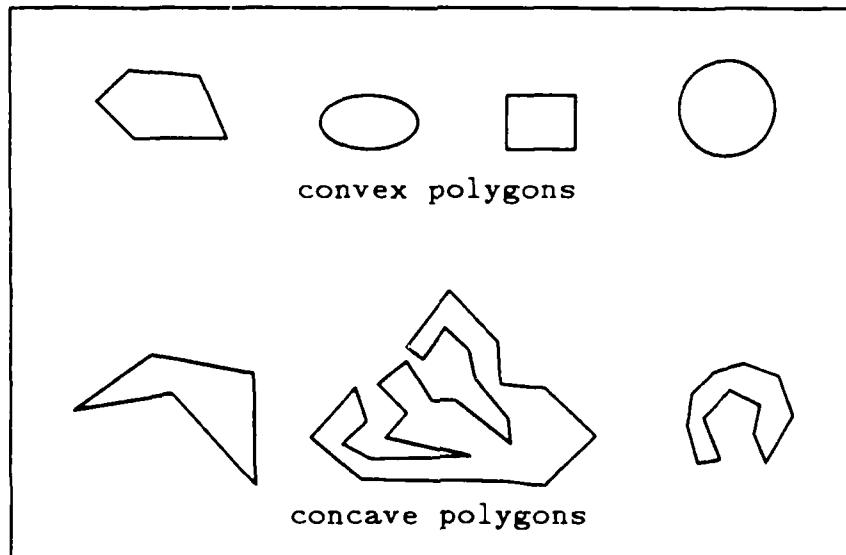
Figure 2.1 - Convex and Concave Polygons
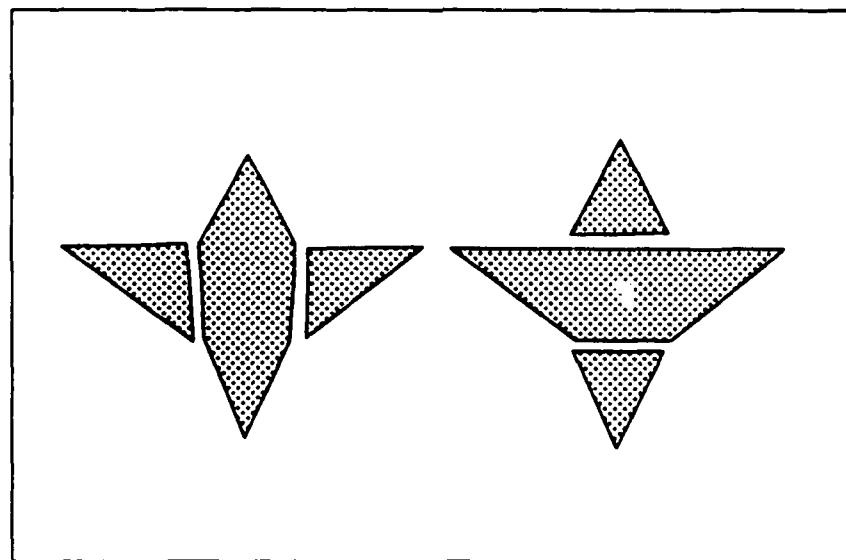
Figure 2.2 - Two Equivalent Decompositions
of a Concave Polygon.

result of two different decompositions of the same polygon should be equivalent as shown in Figure 2.2.

During the operation of decomposing a polygon. there are two main checks that must be conducted to ensure complete and proper decomposition takes place. The first check is to ensure that no area of a created convex polygon falls outside the boundary of the original concave polygon. The second check is to ensure that the concave polygon is completely covered by the total areas of the created convex polygons. Figure 2.3 provides examples of the two checks.

The polygon decomposition algorithm produced for this study is comprised of three basic steps. The first maps the three dimensional coordinates of the polygon into a two dimensional plane. This process generates a polygon that is more easily examined and decomposed. The second step conducts the actual decomposition an performs the two decomposition checks. The third step maps the coordinates of the created convex polygons back to the the original orientation in three-space. At this point. the created convex polygons are then ready for passing to the graphics system's polygon fill hardware.

C. RESEARCH FACILITIES

The IRIS (Integrated Raster Imaging System) 2400 Graphics Workstation. manufactured by Silicon Graphics. Inc., is the target system for our polygon decomposition algorithm. By incorporating custom VLSI chips to replace slower graphics software. the IRIS system can accept polygons in user-defined
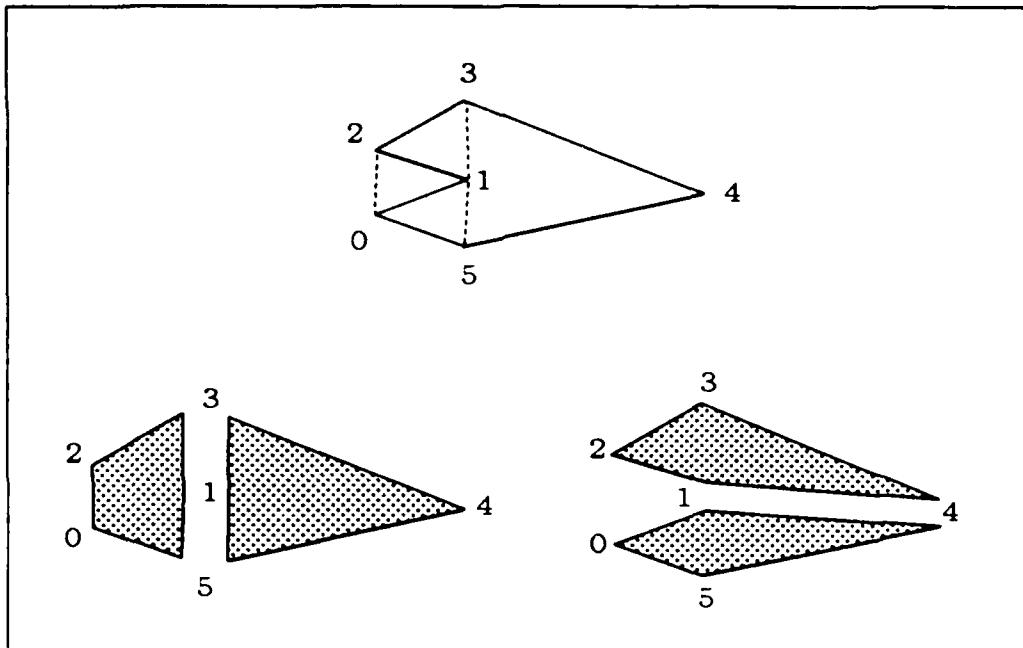
Figure 2.3a - Without a check to ensure the convex
polygon does not fall outside the bounds of the
original polygon, vertices could be chosen that
yield an incorrect decomposition.
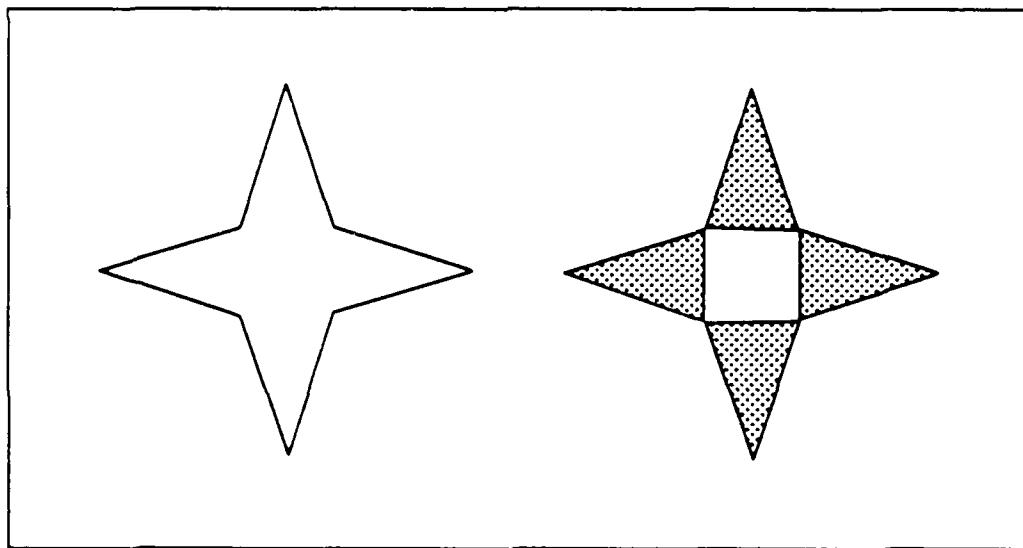
Figure 2.3b - The second decomposition check ensures
the concave polygon is completely covered by the
decomposed polygons. In this case all vertices
and edges were utilized, however a hole is still
left.

coordinates, and transform them to screen coordinates for display on a color graphics monitor that provides 1024 x 768 resolution. This use of special hardware yields processing speeds that are some 100 times faster than similar operations carried out in software.

III. STEP I: MAPPING THE 3D POLYGON TO THE 2D X-Y PLANE

A. THE NORMAL TO A POLYGON

The first step of the polygon decomposition algorithm. is comprised of several small steps. The goal of these steps is to map the polygon to the x-y plane. First. three non-colinear points are selected from the array of x. y. and z coordinates of the polygon. These are used to define the equation of the plane for the 3D polygon. and the normal to that plane (Figure 3.1).

$$Ax + By + Cz = D$$

$$\vec{N} = A_{\vec{i}} + B_{\vec{j}} + C_{\vec{k}}$$

The coefficients A. B. and C are calculated from the coordinates of the three selected points by the following equations:

$$A = (y_0 - y_1)(z_2 - z_1) - (y_2 - y_1)(z_0 - z_1);$$

$$B = (z_0 - z_1)(x_2 - x_1) - (z_2 - z_1)(x_0 - x_1);$$

$$C = (x_0 - x_1)(y_2 - y_1) - (x_2 - x_1)(y_0 - y_1).$$

The coefficients are then used to solve for D of the equation of the plane.
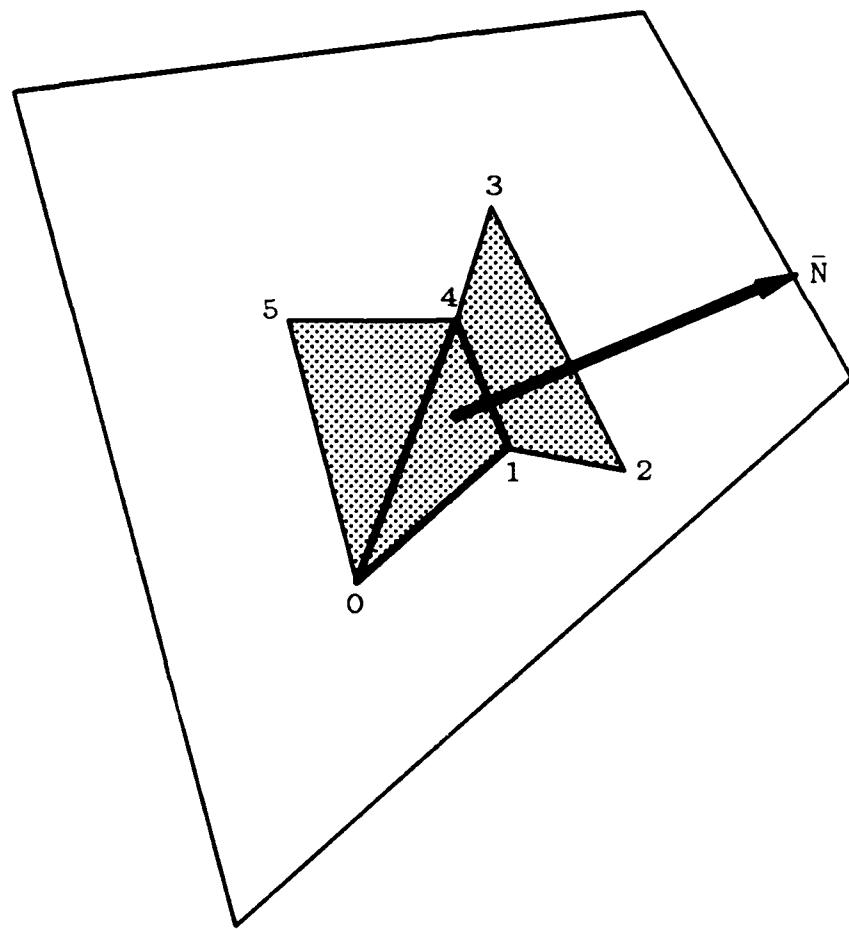
Three vertices 0, 1, and 4
are selected to determine the
Normal to the plane.

Figure 3.1 The Normal to a Polygon

13

B. ANGLES ABOUT THE AXES

Once the equation of the plane is determined. each point in the polygon input array is checked to ensure that the polygon is in fact planar. If the polygon passes this test. then the angles between the normal and the axes of the coordinate system are found. From these angles. the necessary rotations. about the x and y axes. to bring the polygon into the x-y plane. are calculated. The following equations [Ref. 4: pp. 24-59] are used to calculate the angles shown in Figure 3.2a.

$$\cos \alpha = \frac{A}{\sqrt{A^2 + B^2 + C^2}}$$

$$\cos \beta = \frac{B}{\sqrt{A^2 + B^2 + C^2}}$$

$$\cos \gamma = \frac{C}{\sqrt{A^2 + B^2 + C^2}}$$

For the polygon to lie in the x-y plane, the angle between the normal and the z axis is either zero or 180 degrees. Necessarily then rotations are conducted to bring the angles between the normal and the x and y axes to 90 degrees each.

First a projection of the normal onto the x-z plane is made as shown in Figure 3.2b. The dot product of that projected normal N' and the z axis is calculated. The dot product is divided by the product of the magnitudes of the two vectors yielding the cosine of the angle of rotation about the y axis (ϕ).

14
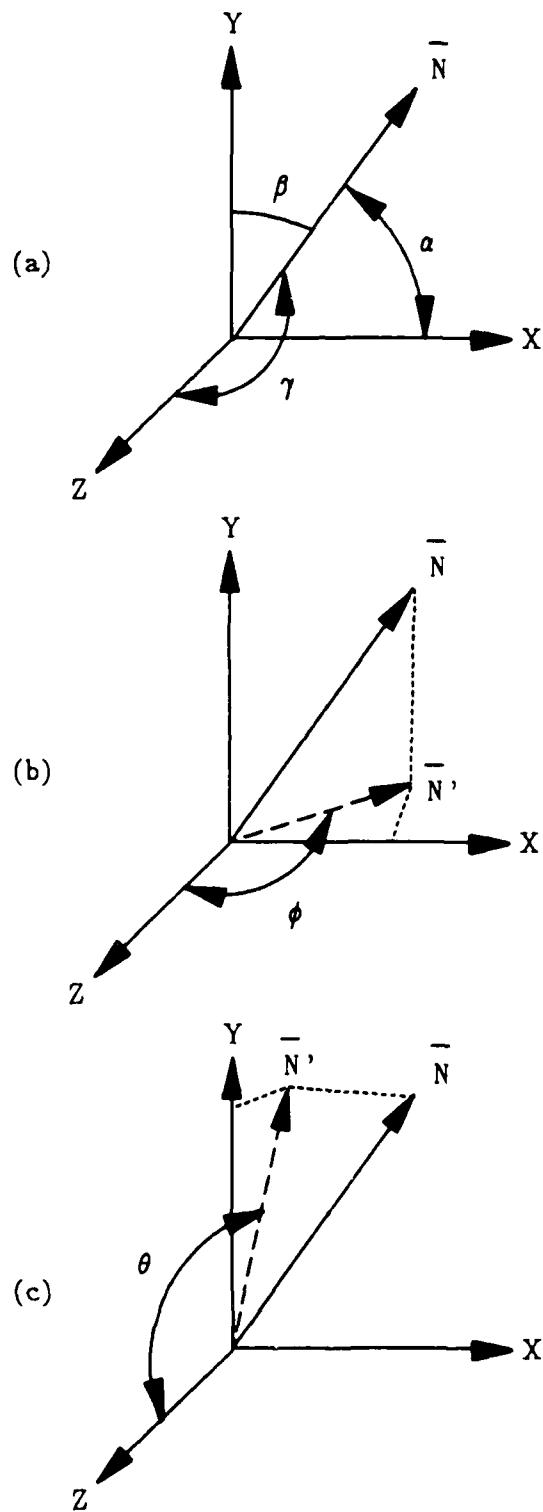
Figure 3.2 Angles about the Axes

15

$$\cos \phi = \vec{N}' \cdot \frac{\vec{Z}}{|\vec{N}'| |\vec{Z}|}$$

A second projection of the normal onto the y-z plane is made and the angle of rotation about the x axis (θ) is calculated in the same manner as above (Figure 3.2c).

C. TRANSFORMATION MATRIX

To actually rotate the polygon into the x-y plane. a transformation matrix must be created that is used in the multiplication of the polygon's coordinates. We first translate the polygon to the origin by adding the negative x. y. and z coordinates of one of the polygon's vertices. Assuming l. m. and n are the coordinates selected. the matrix becomes:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -l & -m & -n & 1 \end{bmatrix}$$

A rotation matrix for the angle ϕ about the y axis is then created

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and concatenated with the translation matrix. This is followed by a multiplication with a rotation matrix for the angle θ about the x axis.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The result of these matrix concatenations is the following transformation matrix that is used to map the array of vertices onto the x-y plane.

$$\begin{bmatrix} \cos\phi & -\sin\phi\sin\theta & -\sin\phi\cos\theta & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ \sin\phi & \cos\phi\sin\theta & \cos\phi\cos\theta & 0 \\ l\cos\phi & n\cos\phi & m\cos\theta+(l\sin\phi-n\cos\phi)(-\sin\theta) & -m\sin\theta+(l\sin\phi-n\cos\phi)(\cos\theta) & 1 \end{bmatrix}$$

Once we have this matrix set up, call it $M_{toplane}$, we then use it to transform the polygon's coordinates via a simple row vector by matrix multiplication. The result of this multiplication is a transformed set of coordinates, in which the z coordinates are zero. The x and y values of this transformed array are then used in the decomposition step of our algorithm. We note at this point that for the mapping of the coordinates back to the original vertices (Step III of the Alg.), that the index into the transformed array is also the index for the untransformed point in the original array of vertices.

17

IV. STEP II: POLYGON DECOMPOSITION

The decomposition step divides the polygon into multiple convex polygons. In so doing, the algorithm performs the two checks to ensure that no area outside the polygon is shaded and that the created convex polygons completely cover the area of the concave polygon. This step requires that the vertices of the polygon be ordered in a clockwise fashion, with respect to the +z directed normal to the x-y plane. Thus a determination of the ordering must be made and if necessary the array of vertices reversed.

A. CLOCKWISE ORDERING

To determine if the vertices of the polygon are ordered clockwise, the polygon's area is calculated using the formula [Ref. 5: p. 369] below. (Note $x, y,$ are the polygon's 2D coordinates.)

$$\frac{1}{2}((x_0y_1) + (x_1y_2) + \cdots + (x_{n-1}y_n) + (x_ny_0) \; (y_0x_1) - (y_1x_2) \; \cdots \; -(y_{n-1}x_n) - (y_nx_0))$$

The above equation yields a negative area for clockwise and a positive area for counterclockwise polygons. If the polygon is ordered counterclockwise then the array of x, y, and z coordinates that define the polygon is reversed before the selection procedure is begun.

B. VERTEX EXAMINATION

The selection of vertices for a convex polygon is conducted by examining each vertex of the concave polygon with respect to the previously selected vertices of the developing convex polygon. This is begun by initially selecting two adjacent vertices. then in clockwise order. examining each of the polygon's remaining vertices to build as large a convex polygon as will fit within the boundary of the concave polygon.

A series of tests are conducted in the selection of vertices for the convex polygon. As vertices are selected. the line segments connecting these become the edges of the convex polygon and define its boundary.

1. Test One

The first test of a vertex is if it lies to the left or right of each line segment of the developing convex polygon. The vertices of the concave polygon are examined in clockwise order. Thus if a vertex lies to the left of any of the line segments of the developing polygon. then a concave angle is necessary to include the vertex in the polygon. As shown in Figure 4.1a. vertex 2 lies to the left of the line segment 0-1. A concave angle (ρ in Figure 4.1b) is necessary to add vertex 2 to the developing polygon. Therefore vertex 2 is not selected and the process moves on to perform the same test on vertex 3.

2. Test Two

A vertex that passes the first test may still require a concave angle to be included in the developing convex polygon. In Figure 4.2. the vertices 0. 1. 2. 3.

19
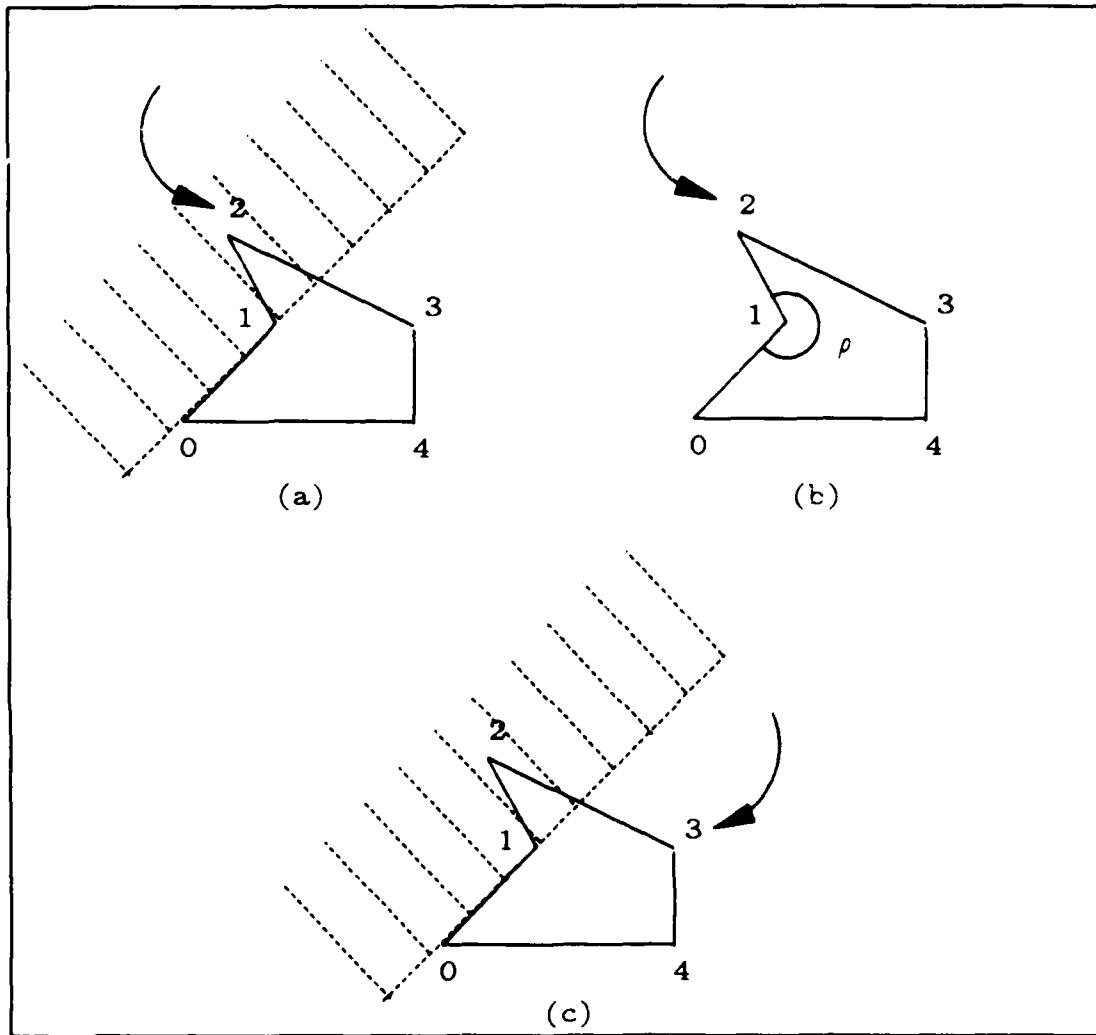
Figure 4.1 - (a) Vertex 2 lies to the left of the line segment 0-1, (b) Concave angle (ρ) would be required to connect Vertex 2 to the developing polygon, (c) Vertex 2 is skipped and the examination is continued with vertex 3.
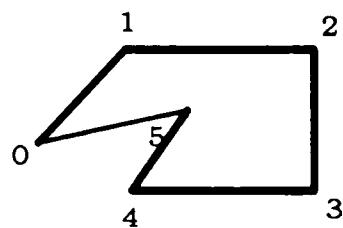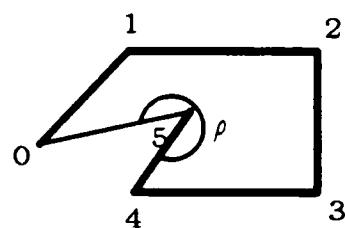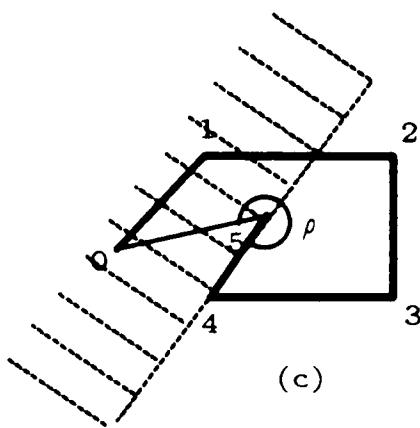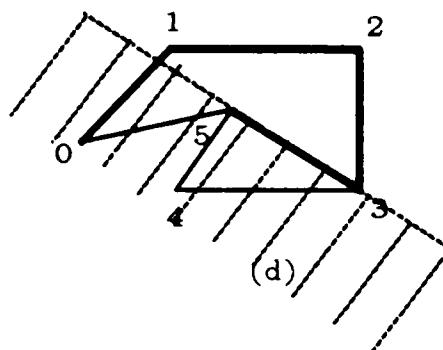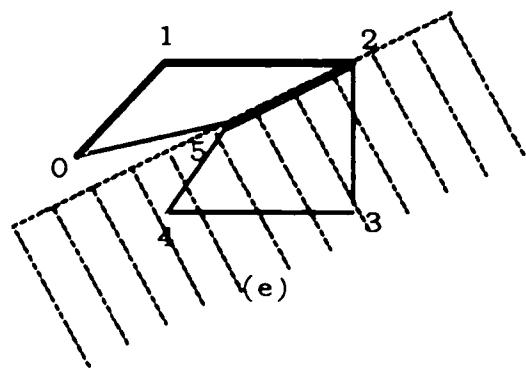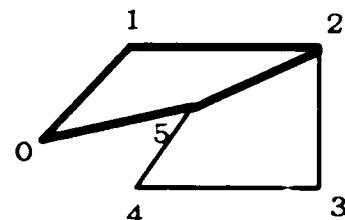
Figure 4.2 - (a) Vertex 5 lies to the right of each line segment of the developing polygon, (b) Concave angle (ρ) is required to connect Vertex 5 to Vertex 0, (c) Concave angle (ρ) can be identified since 0 lies to the left of line segment 4-5, (d) Vertex 4 is skipped and the test repeated with line segment 3-5, (e) Vertex 3 is skipped and the test repeated with line segment 2-5, (f) Result of the tests are that vertices 0, 1, 2 and 5 are selected.

21

and 4 have been previously selected for a convex polygon with vertex 5 under examination. Vertex 5 passes the first test since it lies to the right, or inside of each of the line segments. However, a concave angle (ρ in Figure 4.2b) is again required to connect vertex 5 with vertex 0 of the developing polygon. A concave angle of this type is identified, as shown in Figure 4.2c, if the initial vertex of the developing convex polygon lies to the left of the line segment created by the vertex under examination and the last selected vertex of the developing polygon. In Figure 4.2c, the initial vertex, 0, lies to the left of line segment 4-5. To remedy this, the last selected vertex of the developing convex polygon, vertex 4, is removed and the test is again conducted with the line segment formed by the vertex under examination and the last selected vertex (Figure 4.2d & 4.2e). This process is repeated, removing vertices from the developing polygon, until the initial vertex no longer lies to the left of a new line segment.

3. Test Three

The third test conducted is to ensure that no line segment created in the convex polygon intersects any of the edges of the concave polygon. Such an occurrence results in area outside the concave polygon being included in the convex polygon and an inaccurate polygon fill taking place. In Figure 4.3, the previously selected vertices are 0, 1, 2, and 3. Vertex 7 is under examination and passes the two earlier tests. However, the selection of vertex 7 results in the convex polygon 0, 1, 2, 3, 7 which contains area outside the concave polygon. A situation such as this is identified since line segment 3-7 intersects line segment

22
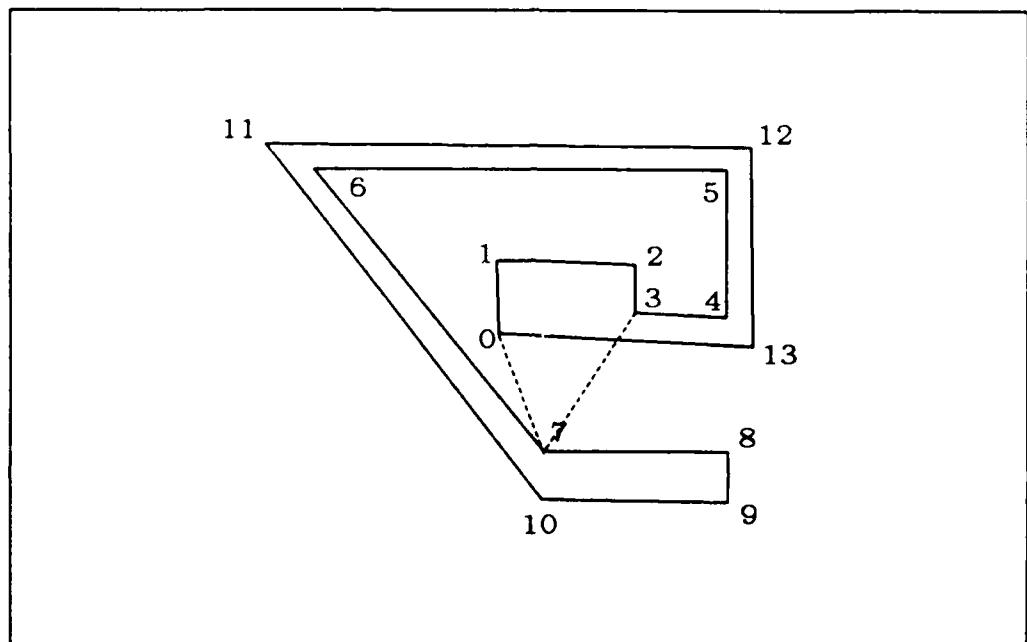
Figure 4.3 – Tests must be conducted for the
intersection of developed line segments with
the edges of the original concave polygon.
Otherwise Vertex 7 would be an acceptable
vertex for inclusion in the developing
polygon 0, 1, 2, 3...

13-0 of the concave polygon. When this occurs, the vertex under examination is removed from the convex polygon and the next vertex of the concave polygon is examined.

C. SELECTION COMPLETION

The selection process for the developing convex polygon concludes when all vertices of the concave polygon have been examined. The convex polygon is then ready for the third step of our algorithm, mapping to the original polygon's 3D coordinates. Figure 4.4 is an example of a concave polygon that was decomposed to the three shaded convex polygons using the above three tests. We have a "hole" left in the concave polygon even though all vertices and edges of the concave polygon have been utilized. Holes such as this are prevented by taking the initial and last vertices of a developed convex polygon as the starting vertices of a convex polygon to be developed. In Figure 4.4, vertices 0 and 2 of polygon 0, 1, 2 are taken to develop convex polygon 0, 2, 4 to fill in the "hole". Once this is accomplished, step two of the algorithm is repeated by selecting a vertex of the concave polygon that has not yet been utilized in a convex polygon. The decomposition process is complete when there are no vertices or edges of the concave polygon that have not been utilized in one or more convex polygons.
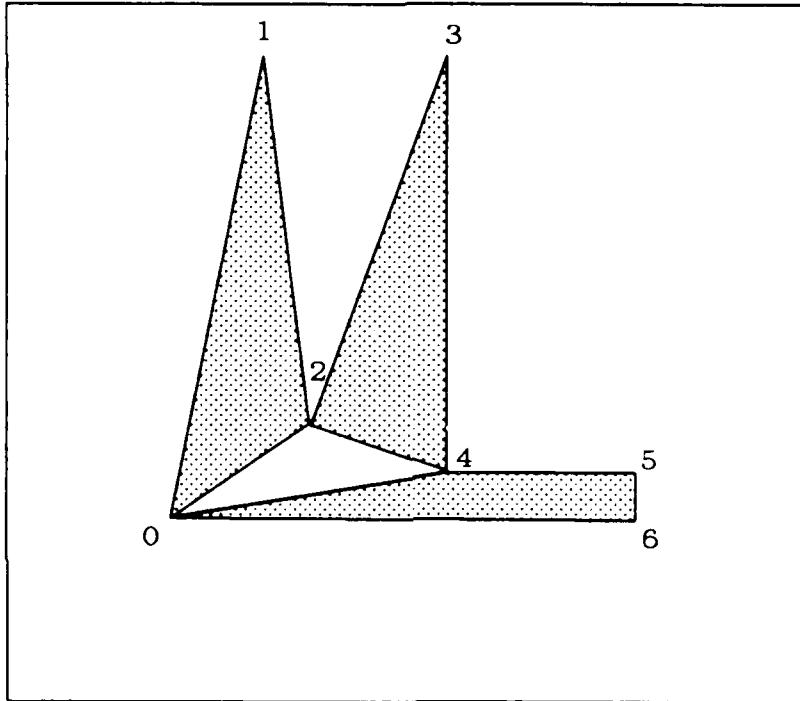
Figure 4.4 - All vertices and edges of
the polygon have been used, yet a hole
remains.

V. STEP III: MAPPING TO THE ORIGINAL COORDINATE SYSTEM

The second and third steps of the decomposition process are interleaved. i.e.. each created convex polygon is immediately mapped back to the original coordinate system. This is accomplished by maintaining the original array of vertices that the user provided for the three-dimensional polygon. As a convex polygon is produced. it is actually represented by the indices of its vertices. Actual vertex coordinates are extracted from the vertex array. as required. using the index. In this manner. the transformed coordinates can be utilized during decomposition and the index can be used to obtain the original coordinates for final display. By this method. there is no consequential need to translate and rotate the vertices back to the original coordinate system.

VI. IRIS IMPLEMENTATION

Implementation of the decomposition algorithm on the IRIS system is via approximately 1600 lines of C code (Appendix). Several data structures and various IRIS system hardware features are utilized as described below.

A. MATRIX IMPLEMENTATION

In the development of the transformation matrix, IRIS system calls for translation and rotation are used to alter the top of the system matrix stack. These routines perform the matrix concatenations needed to translate and rotate the polygon to the x-y plane. To convert the polygon vertices to 2D coordinates, four vertices at a time are loaded into a matrix. The x, y, and z coordinates go into the first three columns with 1's filling the fourth column to provide the necessary homogeneous coordinates. This is placed on the system's matrix stack and multiplied with the transformation matrix using the IRIS's Geometry Engines. The transformed coordinates are removed from the matrix stack and the process is repeated until all vertices have been transformed.

B. DOUBLY LINKED LIST

A data structure for each vertex of the polygon is maintained in our system. This structure contains the vertex's index into the vertex array. As actual coordinates are needed, the vertex's index is used to extract the x, y, and z

27

coordinates from this array. A "used" flag indicates the vertex's use in a developed convex polygon. with a slope and y-intercept stored for the line segment the vertex forms with its predecessor in the array. A forward pointer connects the structure with the next vertex in clockwise order. while a back pointer connects the structure with its preceding vertex. This produces a doubly linked circular list of vertex structures that permits forward clockwise traversal of the polygon and rearward counterclockwise traversal (Figure 6.1).

C. VERTICES STACK

To manage the examination and selection of vertices for a developing convex polygon. a stack is utilized. The size of the vertices stack is determined at run time when contiguous memory is dynamically allocated. based on the number of vertices in the concave polygon. As a vertex is examined. its index is placed on top of the stack along with the slope and y-intercept of the line segment it creates with the vertex below it. If the vertex passes all selection tests. it remains on the stack and a new vertex is pushed on top. On the other hand. when a vertex fails a selection test. it is popped off the stack and the next vertex to be examined is placed on top. Once the selection of vertices for a convex polygon is completed. the vertices are passed to the IRIS system hardware and the stack is cleared to continue with the decomposition process.
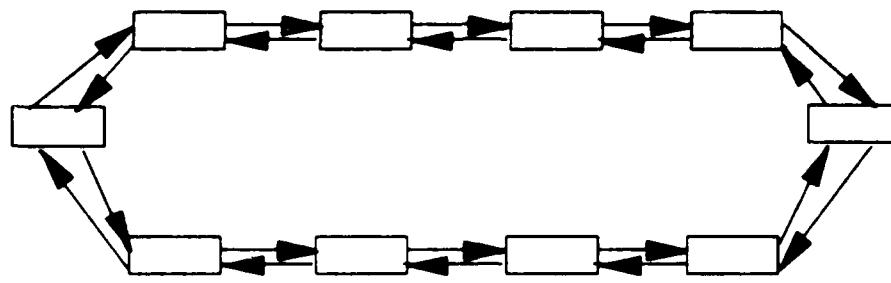
(a) Doubly linked circular list of structures,

Rearward Pointer	Index	Used Flag	Slope	Y-intercept	Forward Pointer

(b) Vertex structure

Figure 6.1

VII. LIMITATIONS AND CONCLUSIONS

This study has presented an algorithm for the decomposition of an arbitrary, planar polygon into a set of convex polygons, thus eliminating the need to manually perform such decomposition. With the algorithm's dynamic allocation of memory, the size polygon that can be filled is limited only by the amount of memory available for use. The algorithm indirectly provides an extended capability over the IRIS system's limit of 384 vertices in one polygon (Ref. 6: p. 2).

However due to the time required to test each vertex for concavity, the algorithm is not suggested for use where real-time response is important. To demonstrate the overhead involved in testing for concavity, a convex polygon was filled and displayed 500 times by the algorithm. The same polygon was then filled utilizing only the IRIS's polygon fill hardware. The results are provided in Figure 7.1 for various polygon sizes.

Due to the three primary steps necessary for decomposition and the tests conducted on each vertex, the speed of the algorithm depends entirely upon the number of vertices and concave angles of the polygon. As such, the success of the algorithm's use in real-time display and animation is based upon a balance between the complexity of the concave polygons used and the speed requirements. We suggest that the created concave polygons be computed once and saved.

30
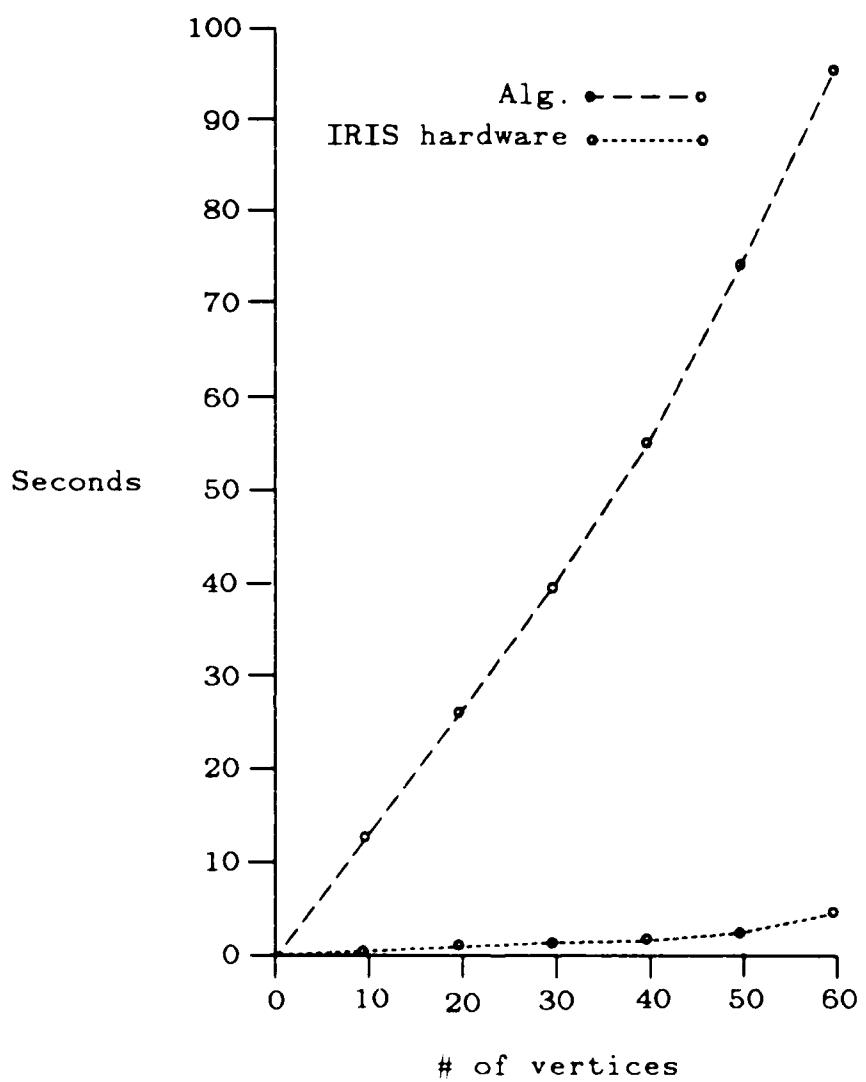
Figure 7.1 - Time required to fill and
display 500 convex polygons with
the number of vertices indicated

31

For IRIS usage, this is perhaps best accomplished through the use of the "object"

mechanism of the IRIS graphics library.

LIST OF REFERENCES

1. Silicon Graphics Inc.. Marketing Brochure. 1984.

2. Pavlidis. Theo. **Structured Pattern Recognition.** Springer-Verlag. 1977.

3. Pavlidis. Theo. **Algorithms** for **Graphics** and **Image Processing.** Computer Science Press. 1972.

4. Rogers. David F.. **Mathematical Elements** for **Computer Graphics.** Kingsport Press. 1976.

5. Shelby. Samuel. **Standard Mathematical Tables.** CRC Press. Inc.. 1974.

6. Silicon Graphics Inc.. **IRIS User's Guide.** Version 2.4. Document Number 5001-051-001-1. 1986.

APPENDIX – ALGORITHM SOURCE CODE

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
            globals.h
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


# include < gl.h >
= include < device.h >
= include < stdio.h >
= include < math.h >


* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
            GLOBAL CONSTANTS
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


# define NULL 0
= define BOTTOM 0
= define  new(x) (x *) malloc(sizeof(x))


* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
            GLOBAL TYPES
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


typedef int boolean;

 * Define a structure to contain data about a vertex of the polygon.
   These structures will be used to create a linked list. *

typedef struct point {
  struct  point *back;
  int     index;
  float   slope;
  Coord   intercept;
  int     direction;
  boolean failed;
  boolean segment_used;
  struct  point *forward;
};

typedef struct point point;
```

/* Define a structure for storing data about each vertex. This will
   be used to create a stack for later use. */

```
typedef struct stack {
  int    index;
  float  slope;
  Coord  intercept;
  int    direction;
};

typedef struct stack Stack;
```

****
GLOBAL DATA STRUCTURES AND VARIABLES
****

Stack *stack;

point *start_ptr, *current_ptr;

int   TOP;

boolean clockwise_error_flag;

```
**************************************************
ROUTINE: convert.c
**************************************************

    This routine takes th  initial array of vertices, passed by
    the user, and makes the calls to other routines to perform the
    necessary translation and rotations to transform the array into
    the x-y-0 plane This new array is then used by the decomoser to
    breakdown the polygon into convex polygons.
**************************************************


cpolf(total pts, pts)

int total pts;
Coord pts  3 ;

{
    int pt1, pt2, pt3, cnt;
    float A, B, C, D, anglex, angley, anglez, 'new pts;
    Matrix transform1;
    Matrix points;

    stack = (Stack ') calloc(total pts - 1, sizeof(Stack));

**************************************************
    dynamically allocate memory to create an array to hold the
    vertices of the newly transformed polygon.
**************************************************


    new pts = (float ') calloc (3'total pts, sizeof(float));

**************************************************
    get 3 non-colinear vertices from the polygon to be used for
    calculating the normal to the polygon.
**************************************************


    extract 3 pts(total pts, pts, &pt1, &pt2, &pt3);

**************************************************
    calculate the coefficients for the equations to the plane
    of the polygon and the normal to that plane.
**************************************************


    determine coeffs(total pts, pts, pt1, pt2, pt3, &A, &B, &C, &D);
```

36

```
*******************************************************
check to ensure all vertices of the polygon lie in the same
plane. If so then calculate the necessary angles of rotation
*******************************************************

if(planar polygon(total pts. pts. A. B. C. D)) {

*******************************************************
calculate the rotation about the y axis necessary to
bring the normal into the y-z plane.
*******************************************************

    angles from normal(A. B. C. 'y'. &anglex. &angley);

*******************************************************
create the transformation matrix necessary to multiply
with the array of vertices to generate the rotation.
*******************************************************

    build transform matrix(transform1. angley. 'y'. pts. pt1);

*******************************************************
perform the matrix multiplication with the array of
vertices and store the new coordinates in the new array
new pts.
*******************************************************

    map pts(total pts. pts. transform1. points. new_pts);

*******************************************************
now extract three non-colinear vertices from the new
array of vertices. new pts.
*******************************************************

    extract 3 pts(total pts. new pts. &pt1. &pt2. &pt3);

*******************************************************
since the polygon has been rotated about the y axis we
calculate the coefficients for the equations to the
polygon one more time.
*******************************************************

    determine coeffs(total pts. new pts. pt1. pt2. pt3. &A. &B. &C. &D);
```

```
/*********************************************/
calculate the rotation about the x axis
/*********************************************/

    angles from normal(A, B, C, 'x', &anglex, &angley);

/*********************************************/
create the transformation matrix
/*********************************************/

    build transform matrix(transform1, anglex, 'x', new pts, pt1);

/*********************************************/
perform the matrix multiplication
/*********************************************/

    map pts(total pts, new pts, transform1, points, new pts);

/*********************************************/
call the routine which will perform the decomposition
upon the newly transformed polygon which now lies in the
x-y-0 plane.(thus to be handled as a 2D polygon)
/*********************************************/

    split(total pts, new pts, pts);

/*********************************************/
once the dynamically allocated memory for the array
new-pts has served its purpose, the memory is freed.
/*********************************************/

    cfree(new pts, 3*total pts, sizeof(float));
    cfree(stack, total pts, sizeof(Stack));

    }

/*********************************************/
if not planar then the decomposition is not attempted.
/*********************************************/

    return;
```

```
*******************************************************************
  ROUTINE: extract.c
*******************************************************************
   This routine searches the array of vertices to find three points which
   do not lie   the same line and returns the indices of these points
*********   ********************************************************


extract_3_pts(total_pts, pts, pt1, pt2, pt3)

int total_pts;
Coord pts[3];
int *pt1, *pt2, *pt3;

{

  int count, finished;

  *pt1   0; *pt2   0; *pt3   0; count   0; finished   0;

   * The first vertex in the array (index  0) is selected for the first point. *

  while(count   total_pts && !finished) {

     * We then search the rest of the array of vertices to locate two more *

    count    1;

    if(*pt2      0) {

       * If the second point has not been selected yet then we check that
         x, y, and z coordinates, of the vertex currently under exam. are
         different. *

      if(pts[*pt1][0] !  pts[count][0]
         pts[*pt1][1] !  pts[count][1]
         pts[*pt1][2] !  pts[count][2] )
         * If so then the vertex becomes the second point. *
        *pt2   count; }

    else if(*pt3     0) {

       * If the third point has not yet been selected, then we check that
         the coordinates of the vertex under examination are different
         from both the first and second point. *

      if((pts[*pt1][0] !  pts[count][0]
          pts[*pt1][1] !  pts[count][1]
```

39

```
            pts *pt1 2  != pts count 2  ) &&
         (pts *pt2 0  != pts count 0
          pts *pt2 1  != pts count 1
          pts *pt2 2  != pts count 2  ))
          '* If the coordinates are different then the vertex is selected
           as the third point. *
         *pt3 = count; }

     else
        finished  1;

     }


     return;

}
```

```
********************************************************
  ROUTINE: is_planar.c
********************************************************

  This routine takes each vertex of the polygon and uses it to solve the
  equation for the plane as calculated earlier.  If every vertex lies
  in the plane, then a value of 1 is returned.
********************************************************

planar polygon(total_pts, pts, A, B, C, D)

int total_pts;
Coord pts[ ][3];
float A, B, C, D;

{

  int count, planar;

  count = 0; planar = 1;

  while(count < total_pts && planar) {

    * The equation for the plane is setup to equal zero if the coordinates
      of the vertex lies in the plane.  A range of .02 on either size of
      zero is allowed to handle round off error. This test is done for
      each vertex in the polygon.  If any vertex fails the test then the
      function returns (0) for non-planar. *

    if((A*(pts[count][0]) + B*(pts[count][1]) + C*(pts[count][2])- D) > .02 ||
       (A*(pts[count][0]) + B*(pts[count][1]) + C*(pts[count][2])- D) < -.02)
      planar = 0;

    count += 1;


  }

  return(planar);

}
```

```
..........................................................
ROUTINE: coeffs.c
..........................................................

This routine uses the three vertices selected from the polygon to
calculate the coefficients that will be used in both the equation
of the plane the polygon lies in and the normal to that plane.
..........................................................


determine_coeffs(total_pts, pts, pt1, pt2, pt3, A, B, C, D)

int     total_pts;
Coord   pts[3];
int     pt1, pt2, pt3;
float   *A, *B, *C, *D;


{

  float x0, x1, x2, y0, y1, y2, z0, z1, z2;;

  x0 = pts[pt1][0];
  y0 = pts[pt1][1];
  z0 = pts[pt1][2];

  x1 = pts[pt2][0];
  y1 = pts[pt2][1];
  z1 = pts[pt2][2];

  x2 = pts[pt3][0];
  y2 = pts[pt3][1];
  z2 = pts[pt3][2];

  *A = (y0-y1)*(z2-z1) - (y2-y1)*(z0-z1);
  *B = (z0-z1)*(x2-x1) - (z2-z1)*(x0-x1);
  *C = (x0-x1)*(y2-y1) - (x2-x1)*(y0-y1);

  *D = -(*A)*pts[pt1][0] - (*B)*pts[pt1][1] - (*C)*pts[pt1][2];

  return;
}
```

```
*********************************************************
ROUTINE: angles.c
*********************************************************

This routine utilizes the coefficients, determined from the normal to
the polygon, to determine the angles of rotation necessary about the x
and y axes.  These rotations are necessary to bring the polygon into the
x-y plane.  This routine is called the first time to calculate the
rotation about the y axis. The second time it is called is to determine
the x axis rotation.  The parameter "axis" is used to pass which angle
is to be calculated.
*********************************************************


angles_from_normal(A, B, C, axis, anglex, angley)

float A, B, C;
char axis;
float *anglex, *angley;


{

  float pi, degx, degy, degz;

  pi = 3.1415926536;

  degx = acos(A/sqrt(A*A+B*B+C*C))*360/(2*pi);
  degy = acos(B/sqrt(A*A+B*B+C*C))*360/(2*pi);
  degz = acos(C/sqrt(A*A+B*B+C*C))*360/(2*pi);

  if(A==0 && B==0) {
     * simply translate the polygon to the origin *
     *anglex = 0.0;
     *angley = 0.0; }

  else if(A==0 && C==0) {
     * rotate polygon 90 degrees about x axis *
     *anglex = 90.0;
     *angley = 0.0; }

  else if(B==0 && C==0) {
     * rotate polygon 90 degrees about y axis *
     *anglex = 0.0;
     *angley = 90.0; }

  else {
     * must calculate amount of rotation about the x & y axes *
     *anglex = acos(C/sqrt(B*B+C*C))*360/(2*pi);
     *angley = acos(C/sqrt(A*A+C*C))*360/(2*pi);
```

43

```
if(axis == 'y') {
  if(degx < 90.0)
    *angley = -(*angley);
  *anglex = 0.0; }
else {
  if(degy   90.0) {
   *anglex = -(*anglex);
   *angley = 0.0; }
 }
}


* convert the angles into tenths of degrees as needed by the IRIS system. *
*anglex   10*(*anglex);
*angley   10*(*angley);

return;
```

```
************************************************************
ROUTINE: transmatx.c
************************************************************

This routine takes a matrix structure and initializes it to an identity
matrix. Then the necessary translation values and rotation values are
added to the matrix to create the resultant transformation matrix.
************************************************************


build transform matrix(transform1, angle, axis, pts, pt1)

Matrix transform1;
float angle;
char axis;
Coord pts 3 ;
int pt1;

{
  int i, j;

    ' create the identity matrix '
  for(i  0; i  4; i   1) {
   for(j  0; j  4; j   1) {
     if(i   j)
       transform1 i j    1;
     else
       transform1 i j    0;
   }
  }
  pushmatrix();
  loadmatrix(transform1);

    ' add in the rotation values '
  rotate((int)angle, axis);

    ' add in the translation values '
  translate(-pts pt1  0 , -pts pt1  1 , -pts pt1  2 );

  getmatrix(transform1);
  popmatrix();

  return;
}
```

45

```
..............................................................
ROUTINE: mapover.c
..............................................................

This routine places up to 4 vertices into a matrix at one time then
performs a matrix multiplication with the transformation matrix to
produce x, y, and z coordinates for the polygon that is translated
and rotated some amount about the x and y axes. These new coordinates
are restored in the array new_pts for further manipulation.
..............................................................


map_p( total_pts, pts, transform1, points, new_pts)

int    total_pts;
Coord  pts[3];
Matrix transform1, points;
float  new_pts[3];



  int count, count2, cnt;

  count = 0; count2 = 0;

  while(count < total_pts)
  {
    cnt = 0;

    while(cnt < 4)
    {
       /* place the coordinates of a vertex into the matrix */
       points[cnt][0] = pts[count][0];
       points[cnt][1] = pts[count][1];
       points[cnt][2] = pts[count][2];
       points[cnt][3] = 1.0;

       if(count < 1 < total_pts)
         cnt += 1;
       else
         break;


    }


       /* save the state of the matrix stack and perform the multiplication */
       pushmatrix();

       loadmatrix(transform1);
       multmatrix(points);
       getmatrix(points
```

```
popmatrix();

cnt = 0;

while(cnt < 4) {
    /* move the newly calculated coordinates back into the vertices array */
    new_pts[count2][0] = points[cnt][0];
    new_pts[count2][1] = points[cnt][1];
    new_pts[count2][2] = points[cnt][2];

    if((count2 + 1) < count)
        cnt += 1;
    else
        break;

}

}

return;

}
```

```
..........................................................
ROUTINE              split.c
..........................................................
    This routine takes the array for the transformed polygon and calls
    the necessary functions and routines to decompose the polygon
    and pass the vertices of the convex polygons to the IRIS system
    for filling
..........................................................


split(total_pts, pts, old_pts)

int   total_pts;
Coord pts[ ][3];
Coord old_pts[ ][3];



    Object  decompose_polygon();
    Object  whole_thing;
    boolean clockwise;
    point   *p1, *p2;

..........................................................
Create the doubly linked circular list, "main chain", for the
concave polygon represented by the pts array.  Also initialize
flags and ptrs that will be used in decomposing the polygon.
..........................................................


    clockwise = clockwise_ordering(total_pts, pts);

    create_main_chain(clockwise, total_pts, pts);

..........................................................
Decompose the main polygon and build the set of simple convex
polygons
..........................................................


    decompose_polygon(total_pts, pts, old_pts);

    return;
```

```
*****************************************************************
ROUTINE:          main_chain.c
*****************************************************************
     Create the doubly linked circular list, "main chain", for the
     concave polygon represented by the pts array.  Also initialize
     flags and ptrs that will be used in decomposing the polygon.
*****************************************************************


create_main_chain(clockwise, total_pts, pts)

boolean clockwise;
int     total_pts;
Coord   pts[3];


{

   int count1;
   point *p1, *p2;

       * allocate memory for a structure to contain data on the first
         vertex of the polygon *


   p1 = new(point);

       * set pointers to this structure and initialize all values *


   start_ptr = p1;
   current_ptr = p1;
   p1->index = 0;
   p1->direction = 0;
   p1->failed = FALSE;
   p1->segment_used = FALSE;

       * if polygon vertices are ordered clockwise then we index the vertices
         array beginning at 0 and incrementing from there.  Other we must begin
         at the other end of the vertices array to extract our coordinates.  In
         this way a forward move along our doubly linked list will equal a
         clockwise move around the polygon. *

   if(clockwise)
      count1 = 1;
   else
      count1 = total_pts-1;

       * for each vertex of the polygon we build a point structure, calculate
         the slope, intercept, initilize flags and connect the structure
         via pointers to its preceeding and succeeding vertices in the
```

49

polygon. The result is a doubly linked list of structures. */

```
do{
  p2 = new(point);
  p1->forward = p2;
  p2->back = p1;
  p1 = p2;
  p1->index = count1;
  determine_slope(pts[p1->back->index][0], pts[p1->back->index][1],
          pts[p1->index][0], pts[p1->index][1], &(p1->slope),
          &(p1->intercept), &(p1->direction));
  p1->failed = FALSE;
  p1->segment_used = FALSE;
  if(clockwise)
    count1 += 1;
  else
    count1 -= 1;
}while(count1 < total_pts && count1 > 0);

/* connect the first vertex structure to the last vertex of the polygon
   and calculate the slope and intercept of the line segment the two
   create. */

p1->forward = start_ptr;
start_ptr->back = p1;
p1 = start_ptr;
determine_slope(pts[p1->back->index][0], pts[p1->back->index][1],
          pts[p1->index][0], pts[p1->index][1], &(p1->slope),
          &(p1->intercept), &(p1->direction));

return;
```

```
**********************************************************
ROUTINE:          clockwis.c
**********************************************************

  This routine calculates the area of the arbitrary planar
  polygon once it has been transformed to the 2D x-y plane.
  A negative area indicates a clockwise ordering of the vertices
  while a positive area indicatesa a negative orderig of vertices
  A "0" is returned for counterclockwise ordering while a "1" is
  returned for clockwise ordering.
**********************************************************


clockwise_ordering(total_pts, pts)

int   total_pts;
Coord pts[3];


{

  float area;
  int   i, result;

  area = 0.0;

  for(i=0; i<total_pts-1; i++) {
    area = area + pts[i][0] * pts[i+1][1];
  }

  area = area + pts[total_pts-1][0] * pts[0][1];

  for(i=0; i<total_pts-1; i++) {
    area = area - pts[i][1] * pts[i+1][0];
  }

  area = area - pts[total_pts-1][1] * pts[0][0];

  if(area >= 0)
    result = 0;
  else
    result = 1;

  return (result);

}
```

```
..............................................................
ROUTINE:              decompose.c
..............................................................

Decompose the main polygon and as convex polygons are determined
these are passed through calls to the IRIS system to be filled.
..............................................................


Object decompose polygon(total pts, pts, old pts)

int   total pts;
Coord pts 3 ;
Coord old pts 3 ;


{


  boolean no point found, search incomplete, adjust needed, OK;
  boolean skip flag, skip search;
  int    pt1, pt2, poly count, value;
  point  'working ptr;

  clear stack();
  get pt(total pts, pts);
  get pt(total pts, pts);

  search incomplete  TRUE;
  skip search  TRUE;
  poly count   0;

  ..............................................................
  Once an initial two points are taken from the concave polygon.
  the process moves through the main chain to select out points
  for constructing the simple convex polygons. Calls are made to
  various routines to check for validity of pts that are accepted
  into a polygon
  ..............................................................

  while(search incomplete) {
   adjust needed  TRUE;
   OK  TRUE;
   while(OK) {
    get pt(total pts, pts);

     * we first check to see if we have made a complete loop of the
      vertices. *

     if(loop completed()) {
      working ptr  start ptr;
```

52

```
    * find the structure in the linked list that matches the
     vertex on top of the stack *

while(working ptr- index ! stack TOP .index) {
 working ptr = working ptr- forward:
}

  * check to see if vertex creates a line segment with the
   vertex below it, on the stack, that intersects any edges
   of the concave polygon *

if(stack TOP-1 .index ! working ptr- back- index) {
 value = intersecting lines(pts stack TOP .index 0 ,
      pts stack TOP .index 1 , pts stack TOP-1 .index 0 ,
      pts stack TOP-1 .index 1 , stack TOP .slope,
      stack TOP .intercept. total pts. pts):
}

if(value) {

  * if vertex creates an intersecting line segment then we
   remove it from the stack and reset the pointer to the
   linked list *

 current ptr = start ptr:
while(current ptr- index ! stack TOP-1 .index) {
 current ptr = current ptr- forward:
 }

current ptr = current ptr- forward:
release pt():
 release pt(): }

else {

  * since the line segment create is OK, we simply remove the
   vertex from the stack since it is the same as the one on
   the bottom of the stack, namely the beginning vertex *

 OK = FALSE:
 release pt():

}

  * if we have not completed the loop then we must conduct the
   tests to ensure valid selection of vertices for a convex
```

53

```
      polygon *

  else if(pt_out_of_bounds(total_pts. pts)) {
    release_pt():
  }
}


 * if at least three points have been selected then... *

if(valid_pts()) {

  * save the beginning and ending vertices for future reference *

  pt1= stack_BOTTOM .index:
  pt2-- stack_TOP .index:

  * pass the vertices to IRIS system hardware for filling and display *

  define_polygon(old_pts):
  poly_count--  1:

   * if this is a normal decomposition and not a "fill in" polygon to
    cover holes left in the concave polygon the the do the following *

  if(skip_search) {

    * set the flags that indicate next time through this loop we
     will be working on a "fill in" polygon *

    adjust_needed   FALSE:
    skip_search- FALSE:

     * adjust the pointers to the linked list so next time through
      the loop we have the start and end vertices of the previous
      polygon to begin working with *

    working_ptr - start_ptr:
    while(working_ptr-.index != pt2) {
      working_ptr- working_ptr- >forward:
    }

     * if the start and end vertices of the previous polygon are
      adjacent on the concave polygon then forget about building
      a "fill in" polygon *

    if(working_ptr- >forward- index !  pt1) {
    while(current_ptr- index !  pt1) {
```

```
          current_ptr = current_ptr->forward:
        }
        get_pt(total_pts, pts):
        current_ptr = working_ptr:
      get_pt(total_pts, pts): }
      else {
      adjust_needed = TRUE:
       }
    }
  }


  ' if the polygon is not fully decomposed yet we select
   two new vertices to continue the decomposition process '

  if(adjust_needed) {
   skip_search = TRUE:
   if(object_complete(total_pts))
    search_incomplete = FALSE:
   else {
    adjust_search_area(pts):
    clear_stack():
    get_pt(total_pts, pts):
    get_pt(total_pts, pts):
   }
  }
 }

return:

}
```

```
..........................................................
ROUTINE:          clear_stack.c
..........................................................

This routine simply resets the stack pointer to -1. In this
manner when the stack has its first item pushed on. the pointer
will accurately reflect a value of 0. (BOTTOM OF STACK)
..........................................................


clear_stack()

{

   TOP = -1;

}
```

```
*******************************************************
 ROUTINE:          get_pt.c
*******************************************************
 This retrieves the next point in the main chain. as identified
 by the current_ptr. and places the point on the top of the stack
 In doing so it increments the TOP variable.  Also as the point
 is retrieved. the slope. intercept. and direction of the line
 segment it creates with the vertex below it on the stack is
 calculated.
*******************************************************


get_pt(total_pts. pts)

int   total_pts:
Coord pts[3]:


{

 TOP + - 1:

 if(TOP - = total_pts) {

    ' select the vertex in the linked list that is currently pointed to '
   stack[TOP].index  current_ptr- index:
    ' and then advance the list pointer '
   current_ptr= current_ptr- forward:
   if(TOP  0) {
     ' take the x. y. and z coordinates for both the vertex on top of
       the stack and the vertex just below it. to calculate the slope
       and y-intercept of the line segment they create '
     determine_slope(pts[stack[TOP-1].index][0]. pts[stack[TOP-1].index][1].
                pts[stack[TOP].index][0]. pts[stack[TOP].index][1].
                &stack[TOP].slope. &stack[TOP].intercept.
                &stack[TOP].direction):
   }
 }
 else {
   printf("STACK overflow occured while retrieving another point");
 }

 return:

}
```

```
.............................................................
ROUTINE:          release_pt.c
.............................................................

This routine removes the item on top of the stack by
decrementing the top of stack pointer TOP.
.............................................................


release_pt()
{

 if(TOP > BOTTOM)
   TOP-=1;
 else {
   printf("STACK underflow occured while trying to release a non-");
   printf("existence point.");
 }


 return;

}
```

```
*********************************************************
ROUTINE:            slopes.c
*********************************************************

     This works through the pts array, and the stack array.  By
     taking value of the TOP of stack and the point beneath it,
     the corresponding coordinates are obtained from the pts array
     and a slope, intercept and direction are calculated.  These
     values are then stored in the stack.
*********************************************************


determine slope(x1, y1, x2, y2, slope, intercept, direction)

Coord x1, y1, x2, y2;
float *slope;
Coord *intercept;
int   *direction;


{

   Coord dx, dy;

    * calculate the delta x and delta y for the to points *

   dx   x2- x1;
   dy   y2- y1;

    * if the line segment is vertical then use the value 99999.0 to
      indicate such (avoiding division by zero problems) *

   if(dx     0) {
     *slope   99999.0;
     *intercept   x2; }
   else {
     *slope   dy  dx;
     *intercept   y2 - (dy  dx) * x2;
   }

    * based upon the changes in delta x and delta y we determine if we are
      traversing down a slope or up a slope *

   if(dy · 0) {
     if(dx · 0)
       *direction   1;
     else
       *direction   0;
   }
```

```
else {
  if(dx   0)
    'direction   0;
  else
    'direction   1;
  :

return;
```

```
***********************************************************
ROUTINE:           compare.c
***********************************************************

This routine determines if a point lies to the left or right of
a line segment. This is done by using the x and y coordinates
of the point with the slope, intercept, and direction of the
line segment to create a parallel line segment. The y intercept
(or in the case of vertical line, the x intercept) is used in
conjunction with the direction of traversal along the line
segment to find which side of the line segment, the point lies
on.
...........................................................


compare_boundry(x1,y1,slope,intercept,direction)

Coord x1, y1;
float slope;
Coord intercept;
int direction;

{

  float test_value;
  int value;

    * if the line segment is vertical then we use the x-intercept to
     determine if a vertex lies to the left or right of another line
     segment *

  if(slope        99999.0)
    test_value    x1;
  else
    test_value    y1- slope * x1;

    * based upon traversal up or down a sloping line we determine if
     a vertex is to the left or right of the line segment *

  switch (direction) {

    case 0: if(test_value    intercept)
          value    1;
        else
        value    0;
        break;

    case 1: if(test_value    intercept)
          value    1;
```

```
        else
          value   0:
        break:
    .

  return (value):
```

```
***************************************************
ROUTINE:          pt_OOB.c
***************************************************

This routine takes the vertex index from the top of the stack
and uses its x, y coordinates to determine if the point lies
out of bounds with respect to the convex polygon being
created.
***************************************************


pt_out_of_bounds(total_pts, pts)

int    total_pts;
Coord pts[3];


{
 Coord  x1, y1;
 int    ptr, value;
 boolean not_finished;
 point  *working_ptr;

   * obtain the coordinates for the vertex on top of the stack *

 x1    pts[stack[TOP].index[0];
 y1    pts[stack[TOP].index[1];
 ptr    TOP-1;
 value  0;

   * use the above coordinates to determine if the vertex lies to
    the left (out of bounds) with respect to any line segment
    created thus far in the selection of vertices *

 while (ptr  BOTTOM && !value) {
  value   compare_boundry(x1,y1,stack[ptr].slope,stack[ptr].intercept,
      stack[ptr].direction);
  ptr-  1;
 }


***************************************************
If point is not out of bounds with any line segments created
then we check to see if the segment produced by this new point
addition causes the start point for the polygon to fall out of
bounds.
***************************************************


 if(!value) {
  x1   pts[stack[BOTTOM].index[0];
  y1   pts[stack[BOTTOM].index[1];
```

```
value= compare  boundry(x1,y1,stack TOP .slope,stack TOP .intercept,
    stack TOP .direction);
if(value) {
  release  pt();
  current  ptr  current ptr- -back;
  working  ptr - start  ptr;
 }
}
```

. . . .

If the newly created line segment does not cause the initial pt
of the developing polygon to fall out of bounds, then we check
to see if the line segment intersects any line segments of the
concave polygon.

. . . .

```
if(!value) {
  working  ptr = start  ptr;
  while(working ptr- -index != stack TOP .index) {
    working  ptr  working ptr- -forward;
  }
  if(stack TOP-1 .index !  working ptr- -back- -index) {

      ' we use the coordinates for the vertex at the top of stack
       and the vertex below it to create a line segment which we
       then check for intersection with any of the edges of the
       original concave polygon '

     value   intersecting  lines(pts stack TOP .index  0 ,
        pts stack TOP .index  1 , pts stack TOP-1 .index  0 ,
        pts stack TOP-1 .index  1 , stack TOP .slope,
        stack TOP .intercept, total  pts, pts);
  }
}
```

. . . .

If the newly created line segment does not intersect any line
segments of the concave polygon then we check to see if the
line segment of the original polygon, that the top of stack
point heads, causes the point below top of stack to fall out.

. . . .

```
if(!value) {
  if(current ptr- -back- -back- -index !   stack TOP-1 .index) {
    x1   pts stack TOP-1 .index  0 ;
    y1 - pts stack TOP-1 .index  1 ;
    value   compare  boundry(x1,y1,current ptr- -back- -slope,
```

```
                    current ptr- back- intercept.
                    current ptr- back- direction):
     }
   }


. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
If the point is determined to be out of bounds. then the
appropriate pointers are adjusted to continue the decomposition
process. Otherwise the pt is a valid addition to the stack
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .


  if(value) {
    working ptr   start ptr:
    not finished  TRUE:
    do {
      if(working ptr- index =   stack TOP .index) {
        not finished  FALSE:
      if(working ptr- failed == FALSE)
        working ptr- failed  TRUE:}
      else {
        working ptr   working ptr- forward:
      }
    } while(not finished):
  }

  return(value):

}
```

```
..................................................................
ROUTINE:          loop_complet.c
..................................................................

This routine checks to see if the vertices of the concave
polygon have been completely traversed while developing a single
convex polygon.  This is done when the vertex on top of the
stack is the same vertex on the bottom of the stack. When this
occurs a value of 1 is returned.
..................................................................


int loop_completed()


    int value;

    if(TOP != -1) {
      if(stack[TOP].index == stack[BOTTOM].index)
        value = 1;
      else {
        value = 0;
      }
    }

    else {
      printf("Error in stack index while checking TOP & BOTTOM values");
      value = -1;
    }


    return (value);
```

```
*****....................................................
ROUTINE:          adjust.c
*****....................................................
   This routine searches the circular list of vertices. checking
   the various condition flags. to locate a vertex or line segment
   of the originial polygon which has not been utilized.  As one is
   located. the list's currency pointer is moved to that structure
   in the list so that further polygon decomposition may continue
...........................................................


adjust search area(pts)

Coord pts  3 :

{

  boolean done:
  point  'p1:

  done   0:
  p1   start ptr:

  do {
    if(p1- failed      1)
       ' check to see if the line segment to either side of the vertex
        has been used in a previously developed polygon. '
      if(!(p1- forward- segment used) !(p1- segment used)) {
       current ptr   p1:
       p1- failed   0:
       done   1: }
      else {
       p1   p1- forward:
       }
     else {
      p1   p1- forward:
     }
  } while(!done && p1 !  start ptr):

  return:

}
```

67

```
*********************************************************
ROUTINE:          valid pts.c
*********************************************************

  This routine checks to ensure that at least 3 vertices have
  been determined for the convex polygon. If not then the
  value of 0 is returned.
*********************************************************


boolean valid pts()
{

  int value;

  if(TOP   2)
    value   1:
  else
    value   0:

  return (value);

}
```

```
**************************************************************
ROUTINE:            define  poly.c
**************************************************************

This routine takes the vertices accumulated on the stack and
uses the IRIS features pmv and pdr to actually perform the
display of the polygon created.  The index for each vertex
is used to access the original array of points to get the x, y
and z coordinates used.
**************************************************************


define  polygon(pts)

Coord  pts  3 :

{

  int   count:
  point  *track  ptr:

  track  ptr   start  ptr:

    * pass the starting vertex for the newly created convex
    polygon to the IRIS hardware with a point move command *

  pmv(pts stack BOTTOM .index  0 .
      pts stack BOTTOM .index  1 .
      pts stack BOTTOM .index  2 ):

    * locate this vertex in the linked list *

  while(track  ptr- ·index !- stack BOTTOM .index) {
    track  ptr   track  ptr- ·forward:
  }

    * set the flag to indicate the vertex has been used
    in a convex polygon *

  if(track  ptr- ·back- ·index      stack TOP .index)
    track  ptr- ·segment used   TRUE:

    * for each vertex of the convex polygon. pass its
    coordinates to IRIS hardware with the point draw
    command *

  count   1:
  while(count ·    TOP) {
    while(track  ptr- ·index !   stack count .index) {
```

69

```
        track_ptr= track_ptr->forward;
     }
    if(track_ptr->back->index == stack[count-1].index)
       track_ptr->segment_used= TRUE;
    pdr(pts[stack_count].index[0],
       pts[stack_count].index[1],
       pts[stack_count].index[2] );
    count -= 1;
  }

   /* pass the command to the IRIS hardware that the
    fill the completed convex polygon */

 pclos();

 clear_stack();

 return;


}
```

```
****************************************************************
ROUTINE:          obj_complet.c
****************************************************************
This routine checks each vertex and edge of the original
concave polygon to see if they have all been utilized in the
process of decomposition. If so a value of 1 is returned.
****************************************************************


int object_complete(total_pts)

int total_pts;

{

  point *p1;
  int    count, value;

  count = 0;
  value = 1;
  p1 = start_ptr;

    /* search through the linked list of structures, for the
       vertices, and return the index of the first vertex that
       creates an edge of the concave polygon not yet utilized */
  do {
    if(p1->segment_used) {
      p1 = p1->forward;
      count += 1; }
    else {
      value = 0;
    }
  } while(count < total_pts && value == 1);

  return (value);

}
```

```
..................................................................
ROUTINE:          intersecting.c
..................................................................

   This routine takes two vertices and the slope and intercept
   of the line segment they create to determine if it intersects
   any of the edges of the original polygon. If so then a value
   of 1 is returned.
..................................................................


intersecting_lines(x2, y2, x1, y1, slope, intercept, total_pts, pts)

Coord x2, y2, x1, y1;
float slope;
Coord intercept;
int   total_pts;
Coord pts[3];


{

   point *p1;
   float x_intersect, y_intersect, x_min, x_max, y_min, y_max, fudge;
   int   value;

      * define a value to offset round off error *

   fudge = 0.000001;

   value = 0;
   p1 =    start_ptr;

      * for each edge of the concave polygon we check if either vertex
        lies on the edge. If not then we determine the intersection
        of the line, defined by the edge, with the line defined by the
        two vertices. Finally we determine if this intersection lies on
        both line segments. If so then we have found an actual intersection. *

   do {

         * check if first vertex lies on the polygon's edge being examined *

      if(((x1 != pts[p1->index][0]) || (y1 != pts[p1->index][1])) &&
         ((x1 != pts[p1->back->index][0]) || (y1 != pts[p1->back->index][1]))) {

            * check if second vertex lies on the edge *

         if(((x2 != pts[p1->index][0]) || (y2 != pts[p1->index][1])) &&
            ((x2 != pts[p1->back->index][0]) || (y2 != pts[p1->back->index][1]))) {
```

72

```
/* if the lines are not parallel then ... */

if(slope != p1->slope) {

    /* if the line created by the vertices is vertical */

    if(slope == 99999.0) {
        /* determine the coordinates of the intersection */
        x_intersect = x2:
        y_intersect = p1->slope * x_intersect + p1->intercept: }

        /* if the edge of the polygon is vertical */

    else if(p1->slope == 99999.0) {
        /* determine the coordinates of the intersection */
        x_intersect = pts[p1->index][0]:
        y_intersect = slope * x_intersect + intercept: }

        /* if neither line is vertical */

    else {
        /* determine the coordinates of the intersection */
        x_intersect = (intercept - (p1->intercept)) / ((p1->slope) - slope):
        y_intersect = slope * x_intersect + intercept:
    }

    /* determine the x range that the intersection must occur in
       to lie on the line segment created by the two vertices */

max_min(x2,x1.&x_max.&x_min):

    /* check if the intersection falls within this range */

if((x_intersect > x_min-fudge)&&(x_intersect < x_max+fudge)) {

        /* since the intersection is within this x range we now determine
           the x range for the edge of the concave polygon */

    max_min(pts[p1->index][0], pts[p1->back->index][0].&x_max.&x_min):

        /* check if the intersection falls within this range */

    if((x_intersect > x_min-fudge)&&(x_intersect < x_max+fudge)) {

            /* now that the intersection falls within the x range of both
               line segments. we must calculate the y range for each
```

73

```
            segment and perform the checks */

        /* y range for the vertices line segment */

        max_min(y2,y1,&y_max,&y_min);

       if((y_intersect >= y_min-fudge)&&(y_intersect <= y_max+fudge)) {

          /* y range for the concave polygon edge */

          max_min(pts[p1->index].1,pts[p1->back->index].1,&y_max,&y_min);
          if((y_intersect >= y_min-fudge)&&(y_intersect <= y_max+fudge)) {
            value = 1;
             /* the intersection exists for the actual line segments and
                therefore the vertex on top of the stack must be removed */
            }
           }
          }
         }
        }
       }
    p1 = p1->forward;
  } while((p1 != start_ptr)&&(value == 0));

  return(value);

}
```

```
.....................................................................
  ROUTINE:          max_min.c
.....................................................................

  Given two float values, p1 and p2, this routine determines
  which is larger of the two.
.....................................................................


max_min(p1, p2, max, min)

Coord p1, p2, *max, *min;

{

  if(p1 < p2) {
    *max = p2;
    *min = p1; }
  else {
    *max = p1;
    *min = p2;
  }

  return;

}
```

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center   2
   Cameron Station
   Alexandria. Virginia 22304-6145

2. Library. Code 0142   2
   Attn: Library. Code 0142
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Chairman. Code 52   1
   Department of Computer Science
   Naval Postgraduate School
   Monterey. California 93943-5000

4. Computer Technology Curricular Officer, Code 37   1
   Naval Postgraduate School
   Monterey. California 93943-5000

5. Dr. Michael J. Zyda. Code 52Zk   2
   Department of Computer Science
   Naval Postgraduate School
   Monterey. California 93943-5000

6. Dr. Robert B. McGhee. Code 52Mz   1
   Department of Computer Science
   Naval Postgraduate School
   Monterey. California 93943-5000

7. Capt Jeffrey H. Potts   8
   C/O Alexander Murphy III
   12701 Skipper Lane
   Hudson, Fla. 33562

END

4-87

DTIC